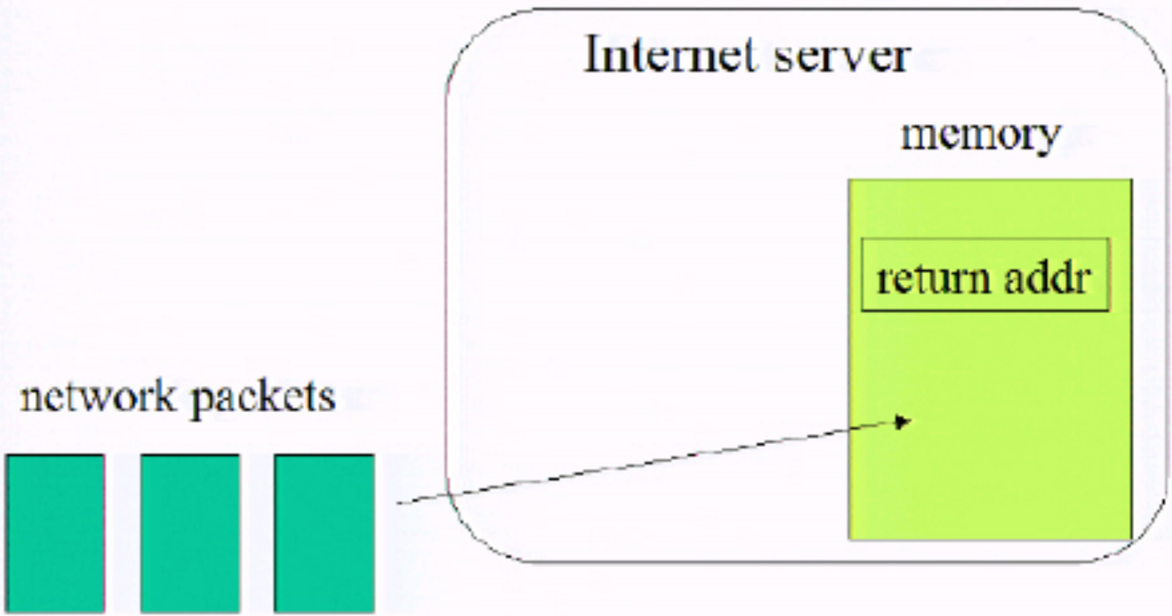
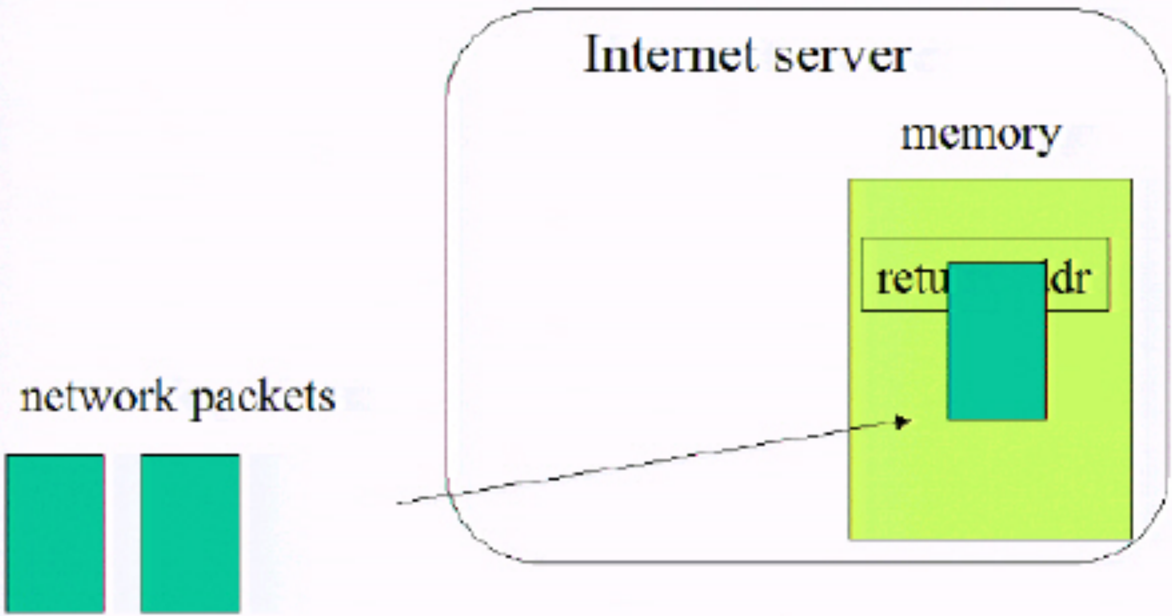
How Low
Can Safe Languages Go?

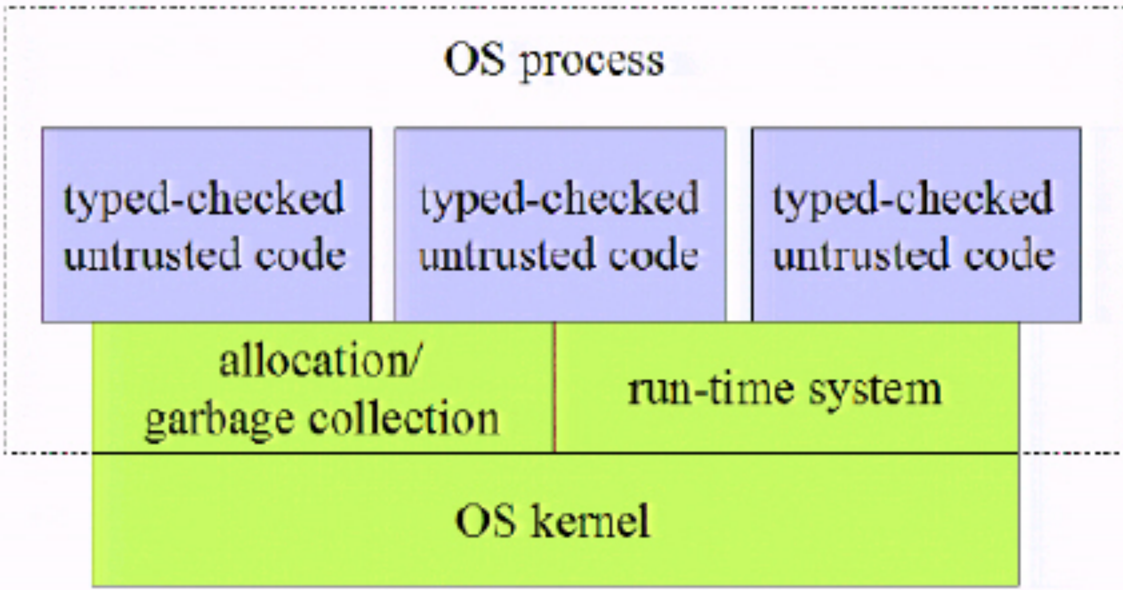
Does safety matter?



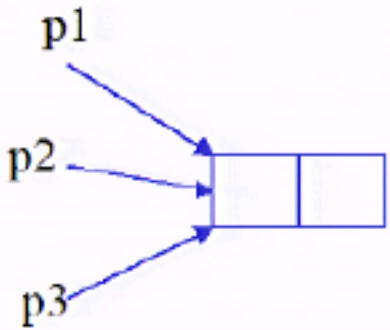
Does safety matter?



Extensible systems



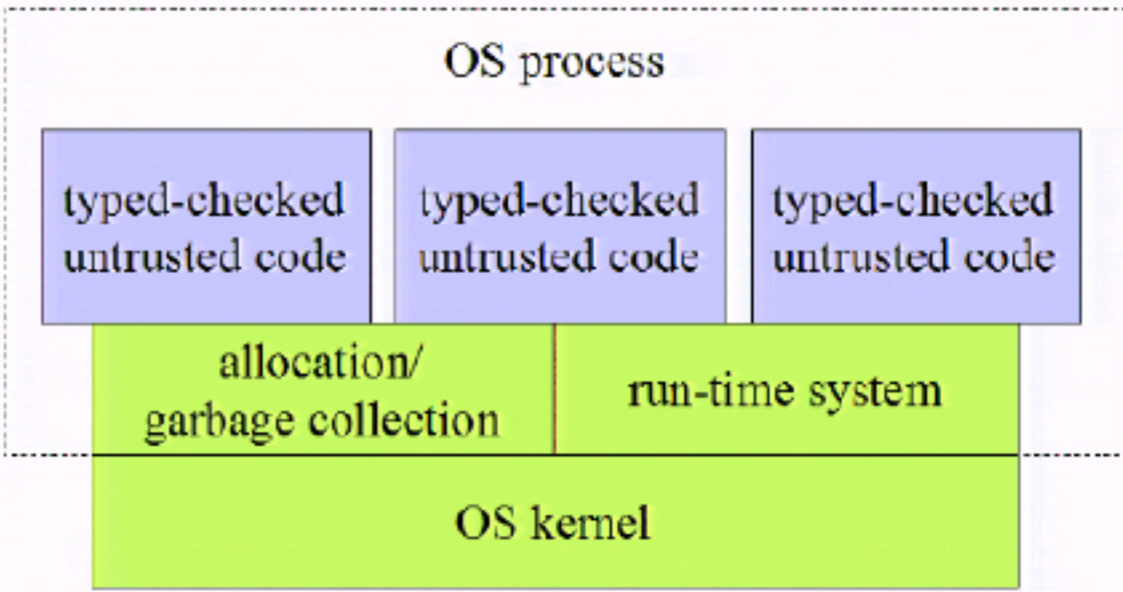
Deallocation and aliasing



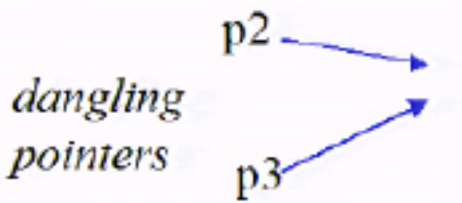

```
free(p1);
```

Safe, unsafe pointers

Extensible systems



Deallocation and aliasing



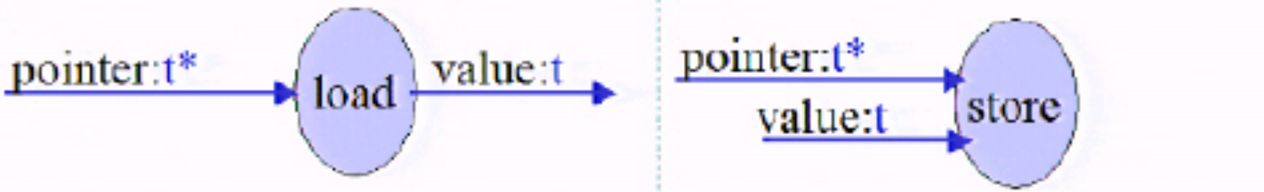
$\boxed{\text{free(p1);}}$

Safe, unsafe pointers

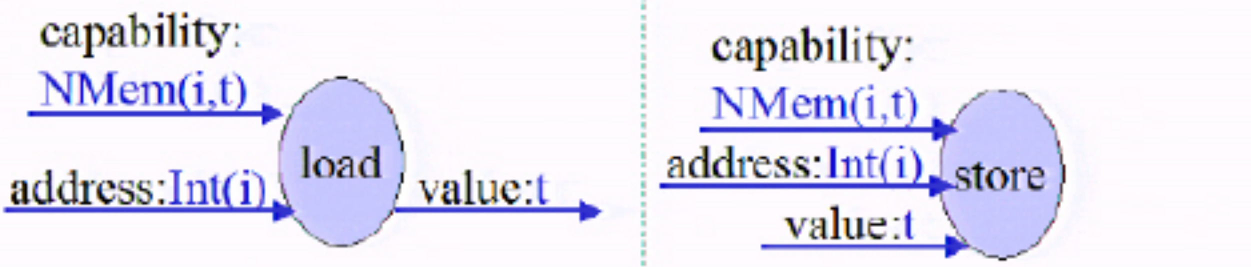
Talk overview

- Static capability types
  - (nonlinear) memory types
  - (nonlinear) device types
- Linear types
  - linear memory types
  - linear device types
  - type-safe garbage collector

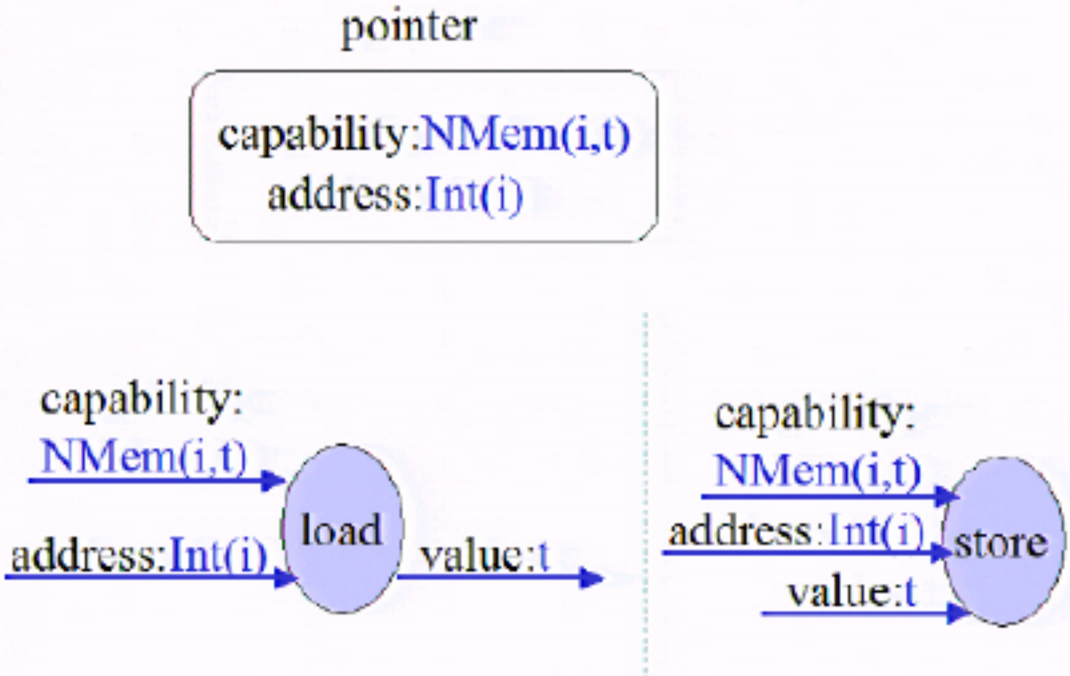
C pointer types (unsafe!)



Nonlinear memory types (safe!)



Pointers



pointer
capability:NMem(i,t)
address:Int(i)
capability:
NMem(i,t)
load
address:Int(i)
value:t
capability:
NMem(i,t)
address:Int(i)
store
value:t

Clay example: pointers

```
type BoolPtr = struct {
  NMem[7,bool] mem;
  Int[4*7] addr;
}
```

pointer

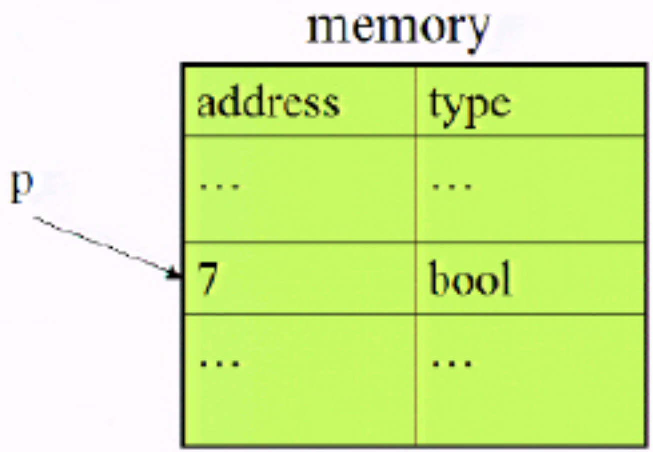
capability:NMem(i,t)
address:Int(i)

memory

address	type
…	…
7	bool
…	…

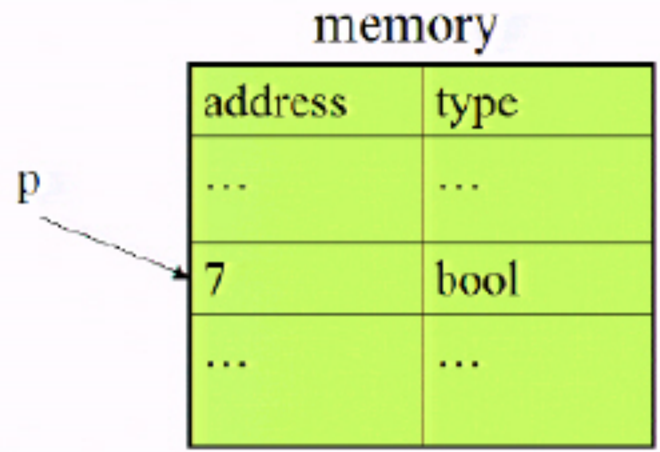
Clay example: pointers

```
type BoolPtr[int I] = struct {
  NMem[I,bool] mem;
  Int[4*I] addr;
}

bool f(BoolPtr[7] p) {
  return g(p, p);
}

bool g[int I1, int I2](BoolPtr[I1] p1, BoolPtr[I2] p2) {
  bool b1 = load(p1.addr, p1.mem);
  bool b2 = load(p2.addr, p2.mem);
  return b1 && b2;
}
```



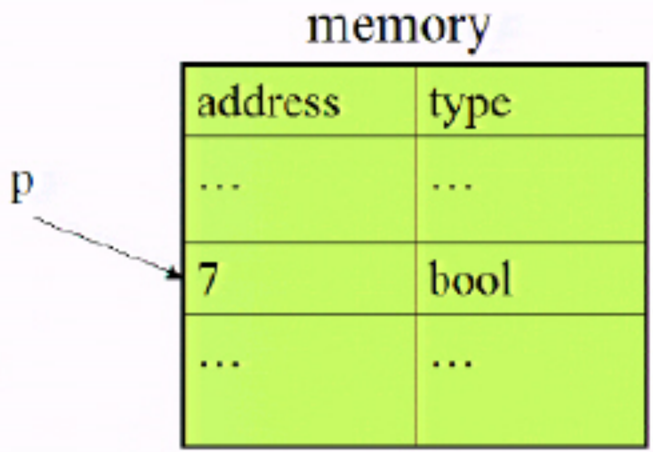
Clay example: null pointers

```
type BoolPtr[int I] = struct {
  NMem[I,bool] mem;
  Int[4*I] addr;
}
```



Clay example: null pointers

```
type BoolPtr[int I] = struct {
  NMem[I,bool] mem;
  Int[4*I] addr;
}

type BoolPtrN[int I] = struct {
  If[I!=0, NMem[I,bool]] mem;
  Int[4*I] addr;
}

bool read[int I](BoolPtrN[I] p) {
  if(p.addr == 0) return false;
  else return load(p.addr, fromIf(p.mem));
}
```



Clay example: pair pointers

```
type BoolPairPtr[int I] = struct {
  NMem[I,bool] mem1;
  NMem[I+1,bool] mem2;
  Int[4*I] addr;
}
```

memory

p

address	type
…	…
7	bool
8	bool
…	…

```
bool h[int I](BoolPairPtr[I] p) {
  bool b1 = load(p.addr,      p.mem1);
  bool b2 = load(p.addr + 4, p.mem2);
  return b1 && b2;
}
```

Clay example: arrays

```
typedef Array[int I, int K, type T] =
  If[I!=K, ArrayNode[I,K,T]]

type0 ArrayNode[int I, int K, type T] =
 struct {  NMem[I,T] mem;
           Array[I+1,K,T] next; }
```

memory

address	type
I	T
I+1	T
I+2	T
…	…
K-1	T

Clay example: arrays

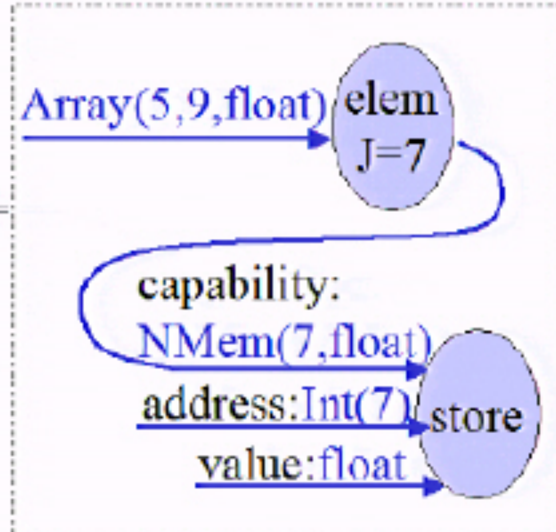
```
typedef Array[int I, int K, type T] =
   If[I!=K, ArrayNode[I,K,T]]

type0 ArrayNode[int I, int K, type T] =
  struct {  NMem[I,T] mem;
            Array[I+1,K,T] next; }

NMem[J,T]
   elem[int I, int J, int K, type T;
         I<=J && J<K](Array[I,K,T] arr) proof[K-I]
  {
     if[I==J] return fromIf(arr).mem;
     else return elem[J=J](fromIf(arr).next);
  }
```

memory

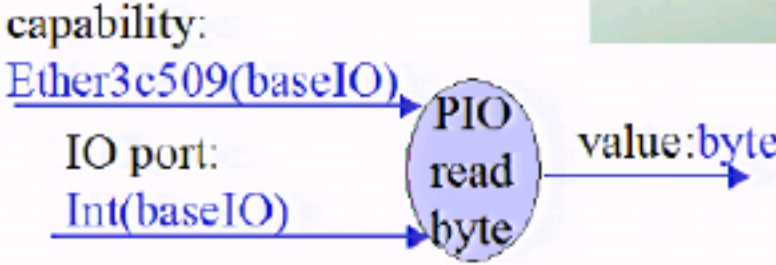
address	type
I	T
I+1	T
I+2	T
…	…
K-1	T

Clay example: arrays

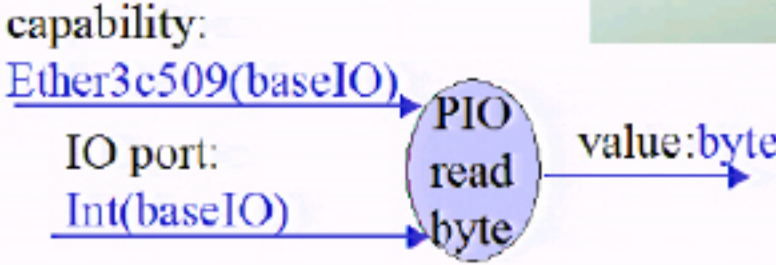
```
typedef Array[int I, int K, type T] =
   If[I!=K, ArrayNode[I,K,T]]

type0 ArrayNode[int I, int K, type T] =
  struct { NMem[I,T] mem;
           Array[I+1,K,T] next; }

NMem[J,T]
   elem[int I, int J, int K, type T;
          I<=J && J<K](Array[I,K,T] arr) proof[K-I]
   {
      if[I==J] return fromIf(arr).mem;
      else return elem[J=J](fromIf(arr).next);
   }
```



Clay example: devices



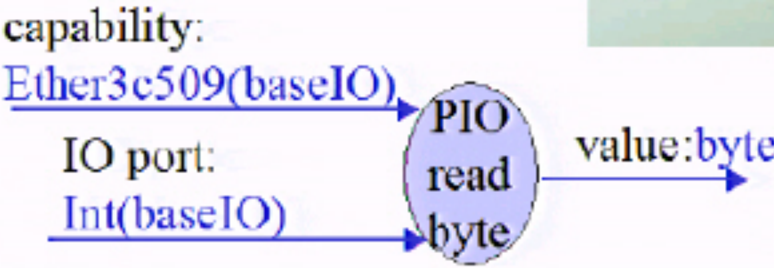
Clay example: devices



…but how do we know that data is ready?

…and that the device is configured to receive?

Clay example: devices



Clay example: devices

capability:
Ether3c509(baseIO)

IO port:
Int(baseIO)

PIO
read
byte

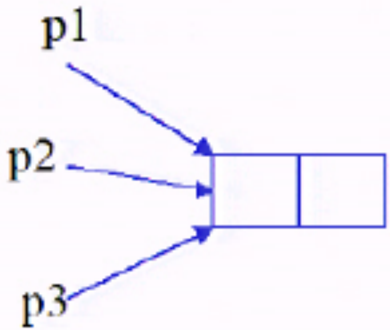
value:byte

…but how do we know that data is ready?
…and that the device is configured to receive?

Talk overview

- Static capability types
  - (nonlinear) memory types
  - (nonlinear) device types
- Linear types
  - linear memory types
  - linear device types
  - type-safe garbage collector

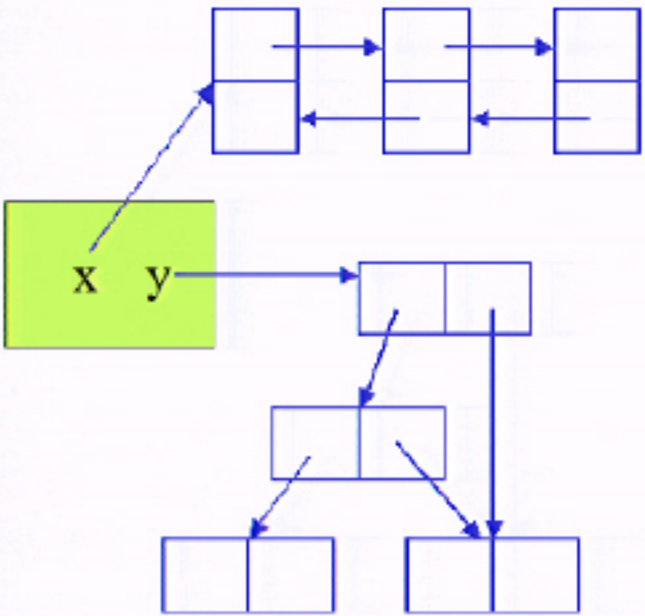
Deallocation and aliasing




```
free(p1);
```

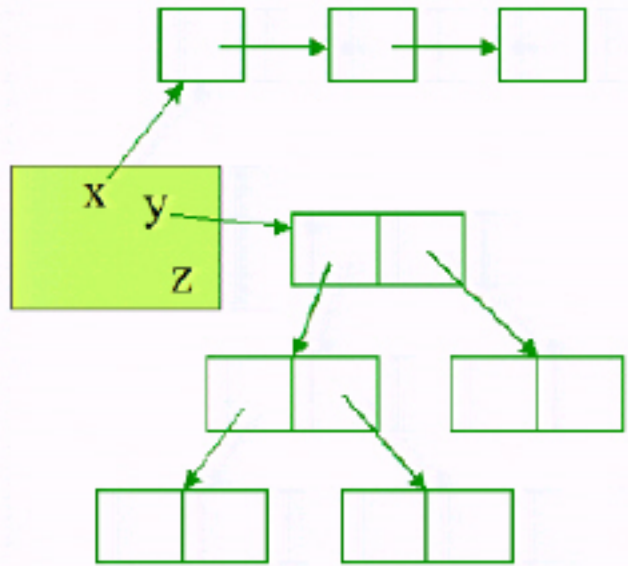
Ban aliasing???

aliasing allowed



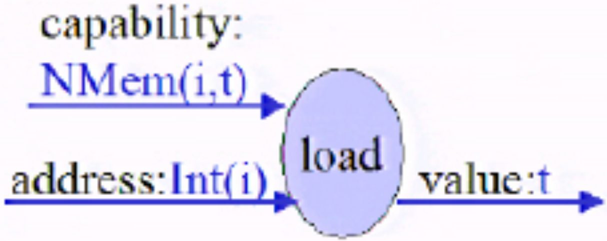
“nonlinear data”

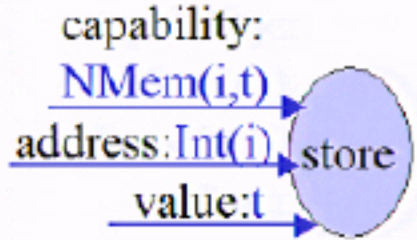
aliasing disallowed



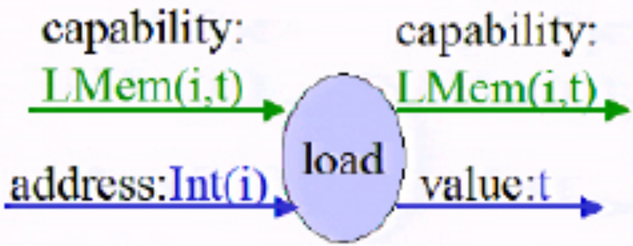
“linear data”

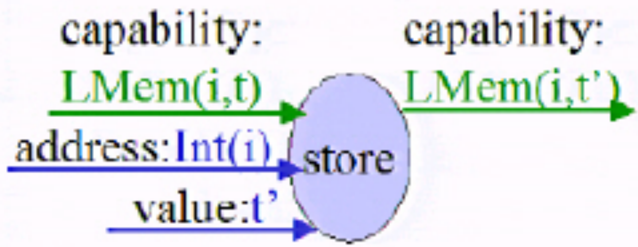
Nonlinear memory types



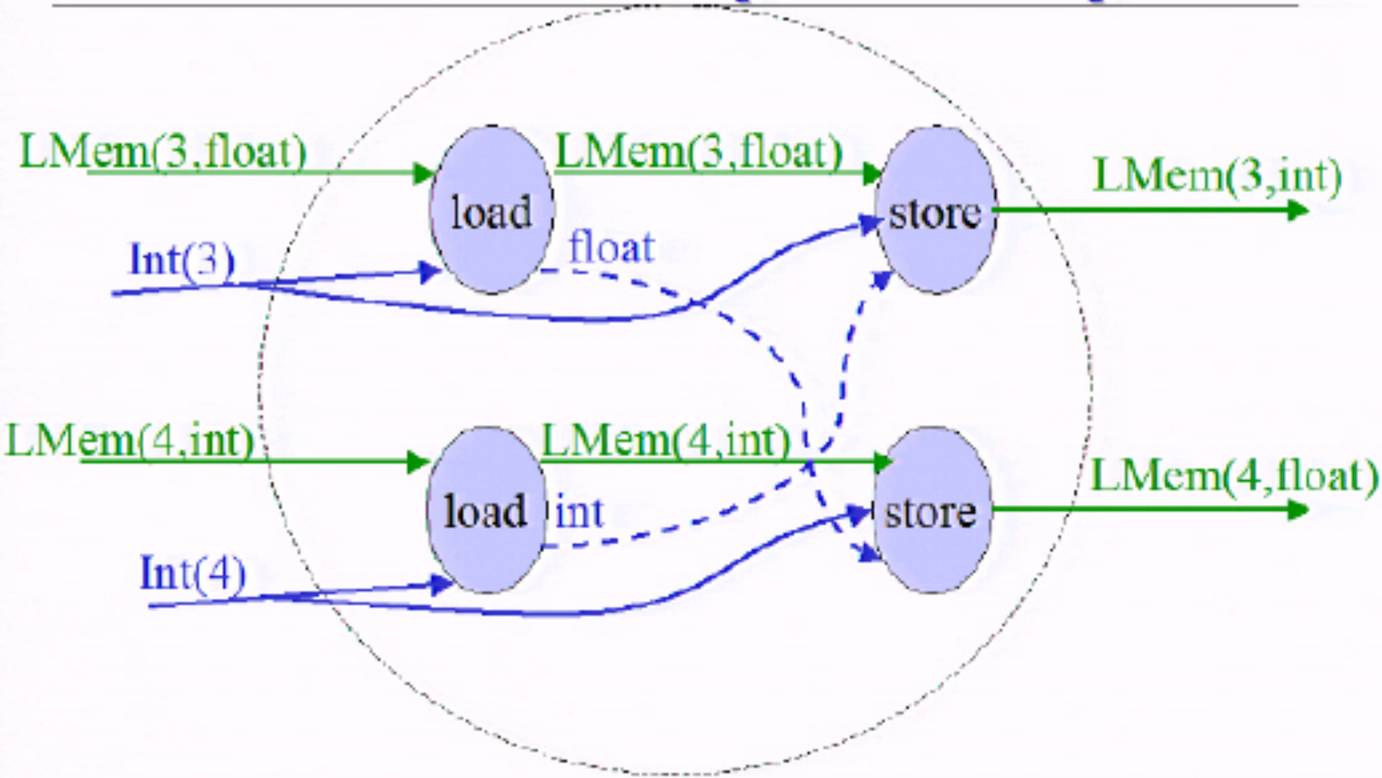


Linear memory types





Linear example: swap



Linear example: devices

capability:
Ether3c509(baseIO,rWin,
txFree,rxAvail)
where rWin==1 && rxAvail>0

PIO
read
byte

capability:
Ether3c509(baseIO,rWin,
txFree,rxAvail-1)

IO port: Int(baseIO)

value:byte

(joint work with Lea Wittie)

Linear example: devices

capability:
Ether3c509(baseIO,rWin,
txFree,rxAvail)
where rWin==1 && rxAvail>0

IO port: Int(baseIO)

PIO read byte

capability:
Ether3c509(baseIO,rWin,
txFree,rxAvail-1)

value:byte

capability:
Ether3c509(baseIO,rWin,
txFree,rxAvail)

IO port: Int(baseIO+14)

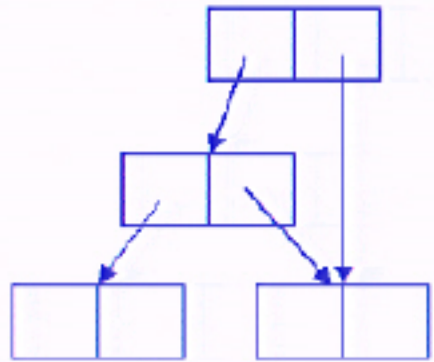
set reg-window command:
Int(0x0800+rWin') where 0<=rWin'<=7

set reg win

capability:
Ether3c509(baseIO,rWin',
txFree,rxAvail)

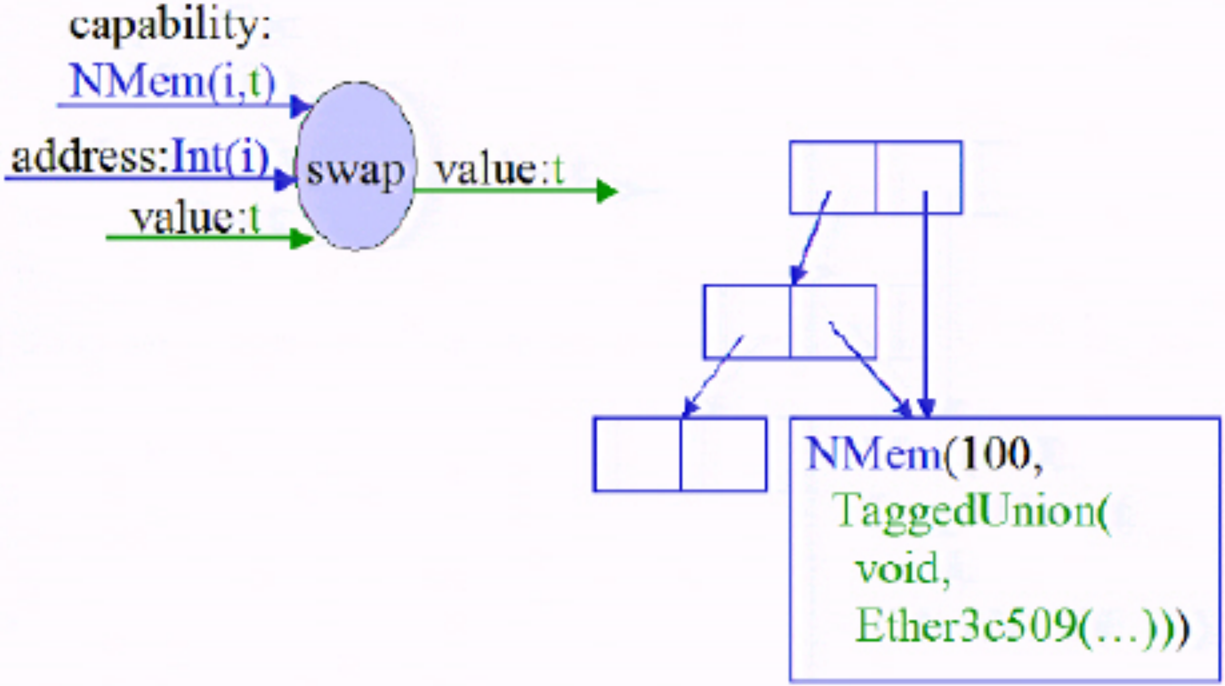
(joint work with Lea Wittie)

Programmers need aliasing!



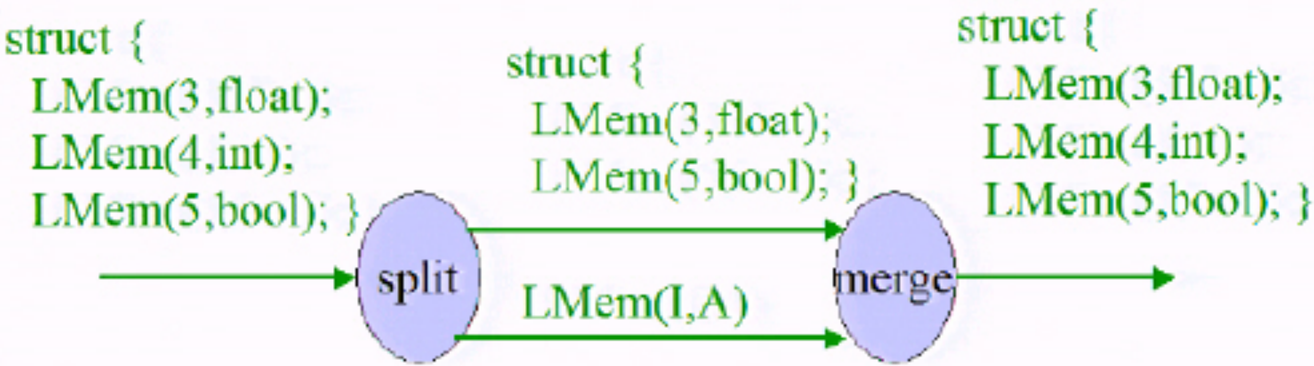
- Dynamic checking
  - Locks
- Static checking
  - Alias types, capability calculus [Crary,Morrisett,Smith,Walker]
  - Linear pools
    - Encode alias types, capability calculus *(work in progress)*
    - Implement garbage collector safely

Locks

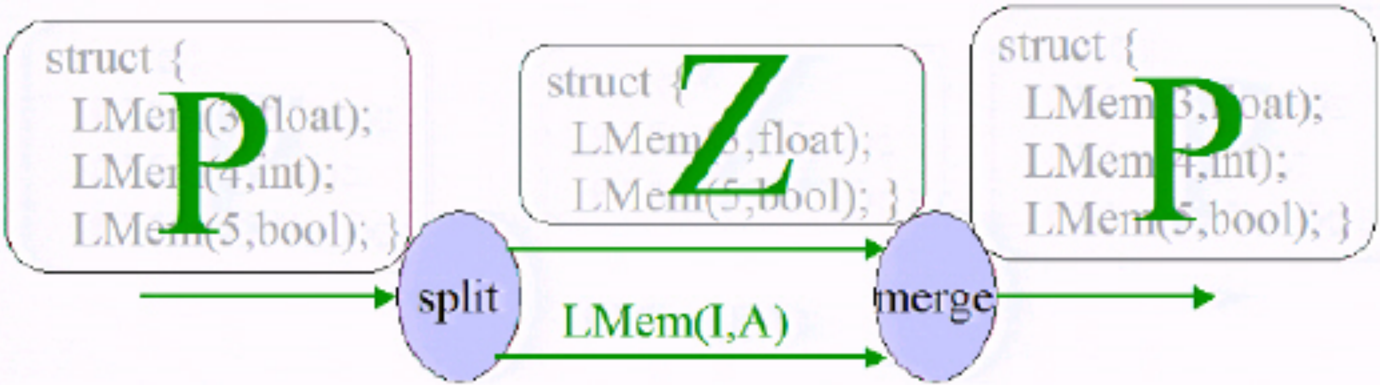


(thanks to Dan Grossman)

Linear pools



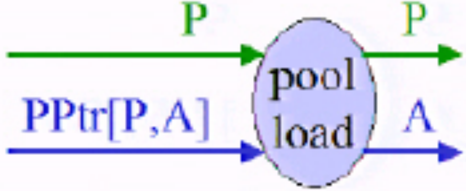
Linear pools




```
type PoolPtr[@type0 P, type A, int I, @type0 Z] = struct {
  Int[I] addr;
  (@[Z,LMem[I,A]]<-proof(P)) split;
  (P<-proof(Z,LMem[I,A])) merge;
}
typedef PPtr[@type0 P, type A] =
   exists[int I, @type0 Z] PoolPtr[P,A,I,Z]
```

Linear pools

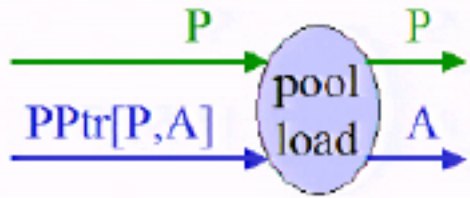
```
inline @[P,A] pLoad[@type0 P, type A](P pool, PPtr[P,A] ptr) {
  let[] (addr, split, merge) = ptr;
  let (z, mem) = split(pool);
  let (a, mem) = load(addr, mem);
  let pool = merge(z, mem);
  return @(pool, a);
}
```




```
type PoolPtr[@type0 P, type A, int I, @type0 Z] = struct {
  Int[I] addr;
  (@[Z,LMem[I,A]]<-proof(P)) split;
  (P<-proof(Z,LMem[I,A])) merge;
}
typedef PPtr[@type0 P, type A] =
   exists[int I, @type0 Z] PoolPtr[P,A,I,Z]
```

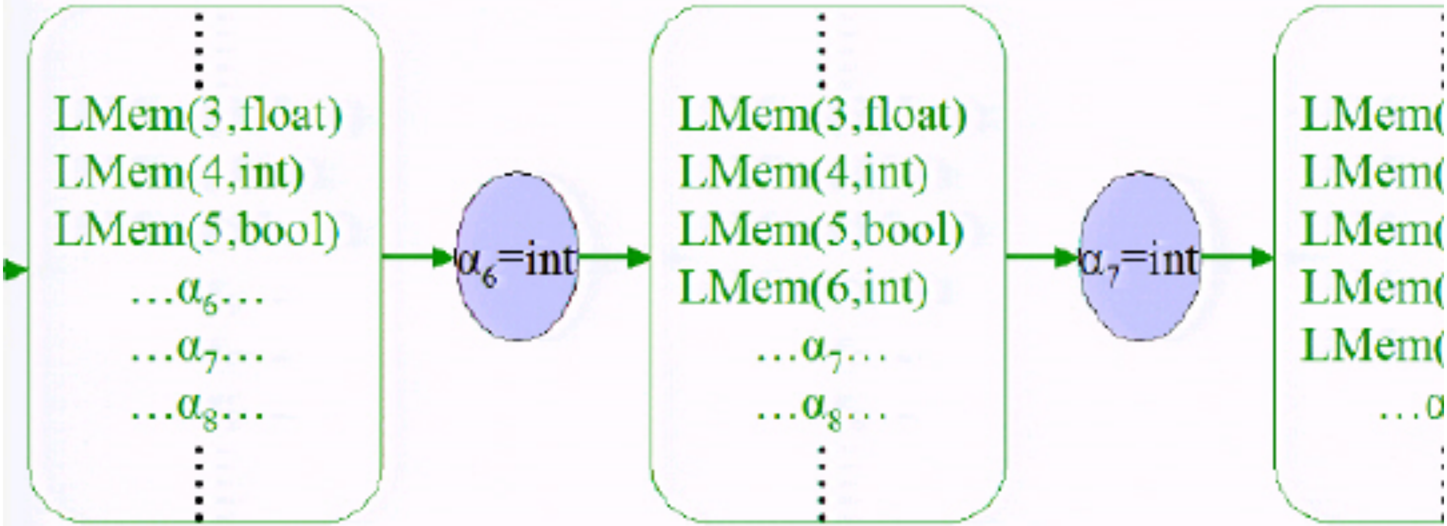
Linear pools

```
inline @[P,A] pLoad[@type0 P, type A](P pool, PPtr[P,A] ptr) {
  let[] (addr, split, merge) = ptr;
  let (z, mem) = split(pool);
  let (a, mem) = load(addr, mem);
  let pool = merge(z, mem);
  return @(pool, a);
}
```



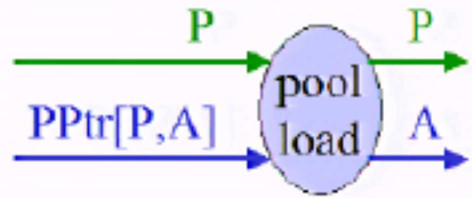

```
@[P,bool] f[@type0 P](P pool, PPtr[P,bool] p) {
 return g(pool, p, p);
}
@[P,bool] g[@type0 P](P pool, PPtr[P,bool] p1, PPtr[P,bool] p2) {
 let (pool, b1) = pLoad(pool, p1);
 let (pool, b2) = pLoad(pool, p2);
 return @(pool, b1 && b2);
}
```

Pools → Regions



Linear pools

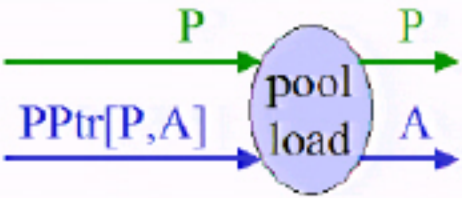
```
inline @[P,A] pLoad[@type0 P, type A](P pool, PPtr[P,A] ptr) {
  let[] (addr, split, merge) = ptr;
  let (z, mem) = split(pool);
  let (a, mem) = load(addr, mem);
  let pool = merge(z, mem);
  return @(pool, a);
}
```




```
@[P,bool] f[@type0 P](P pool, PPtr[P,bool] p) {
  return g(pool, p, p);
}
@[P,bool] g[@type0 P](P pool, PPtr[P,bool] p1, PPtr[P,bool] p2) {
  let (pool, b1) = pLoad(pool, p1);
  let (pool, b2) = pLoad(pool, p2);
  return @(pool, b1 && b2);
}
```

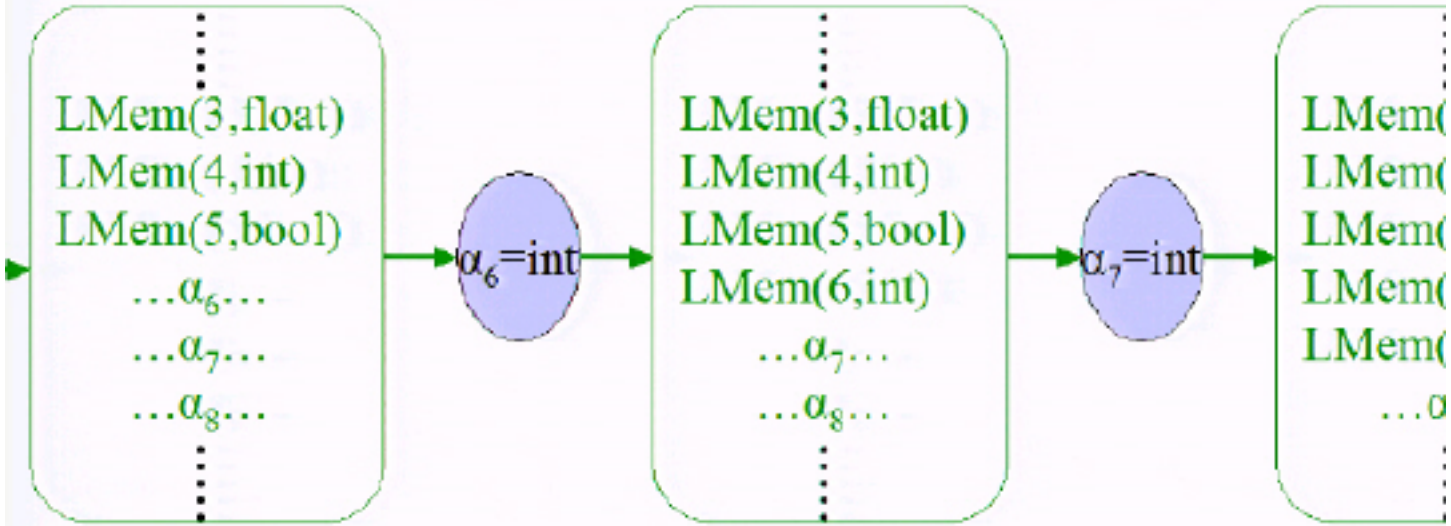
Linear pools

```
inline @[P,A] pLoad[@type0 P, type A](P pool, PPtr[P,A] ptr) {
  let[] (addr, split, merge) = ptr;
  let (z, mem) = split(pool);
  let (a, mem) = load(addr, mem);
  let pool = merge(z, mem);
  return @(pool, a);
}
```

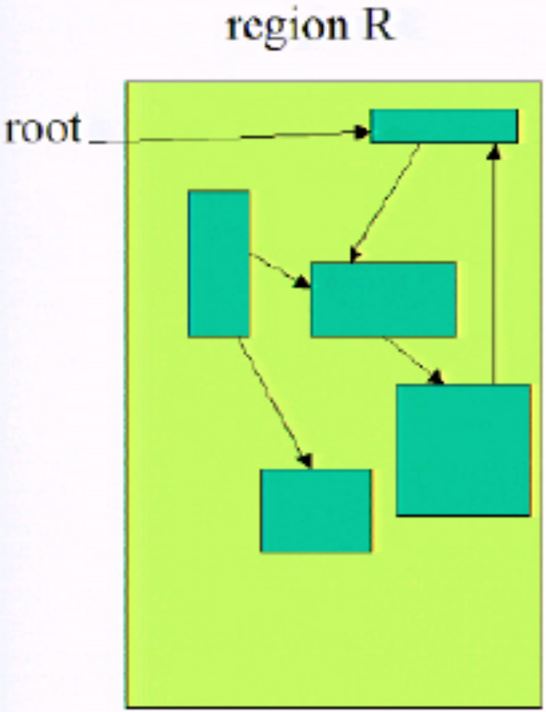



```
type PoolPtr[@type0 P, type A, int I, @type0 Z] = struct {
  Int[I] addr;
  (@[Z,LMem[I,A]]<-proof(P)) split;
  (P<-proof(Z,LMem[I,A])) merge;
}
typedef PPtr[@type0 P, type A] =
   exists[int I, @type0 Z] PoolPtr[P,A,I,Z]
```

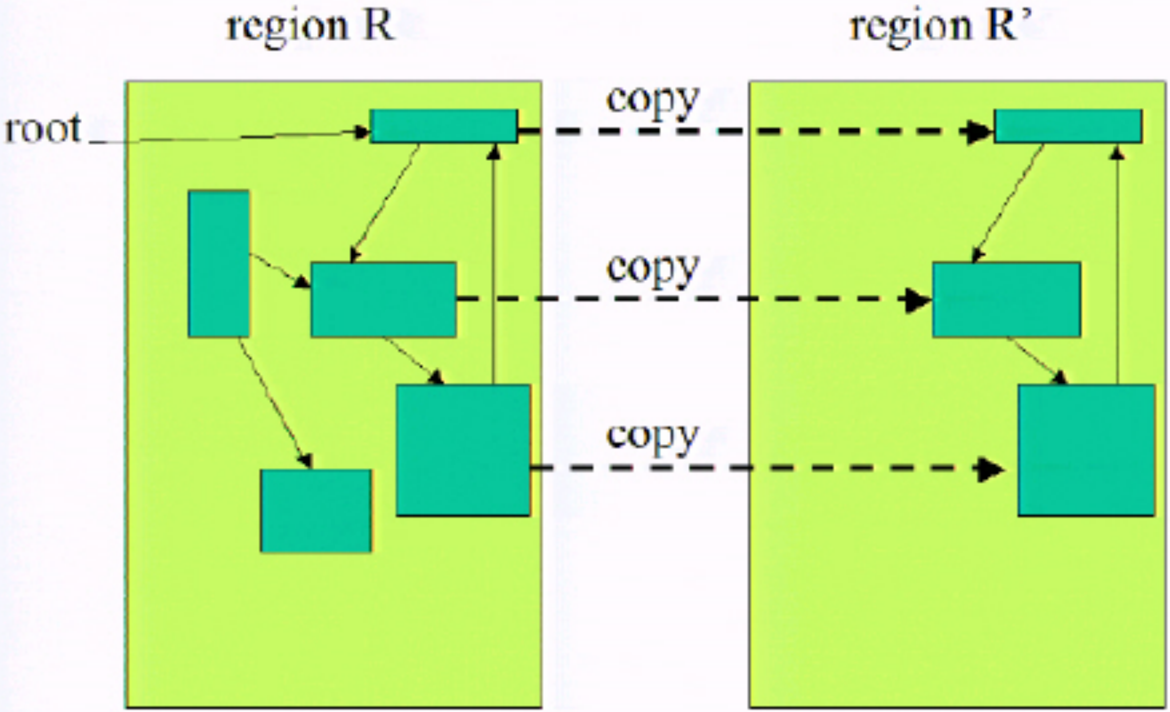
Pools → Regions



Regions → Copying GC

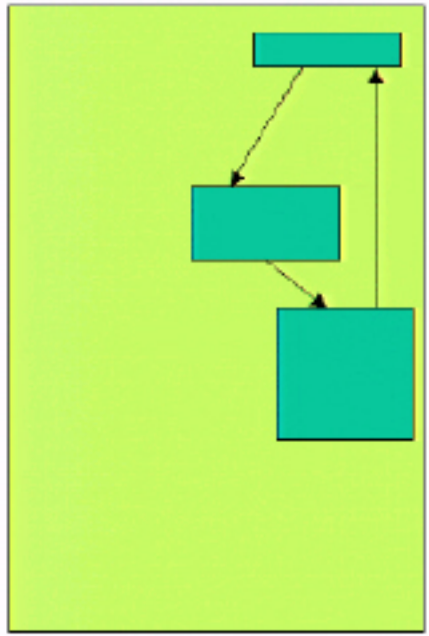


Regions → Copying GC

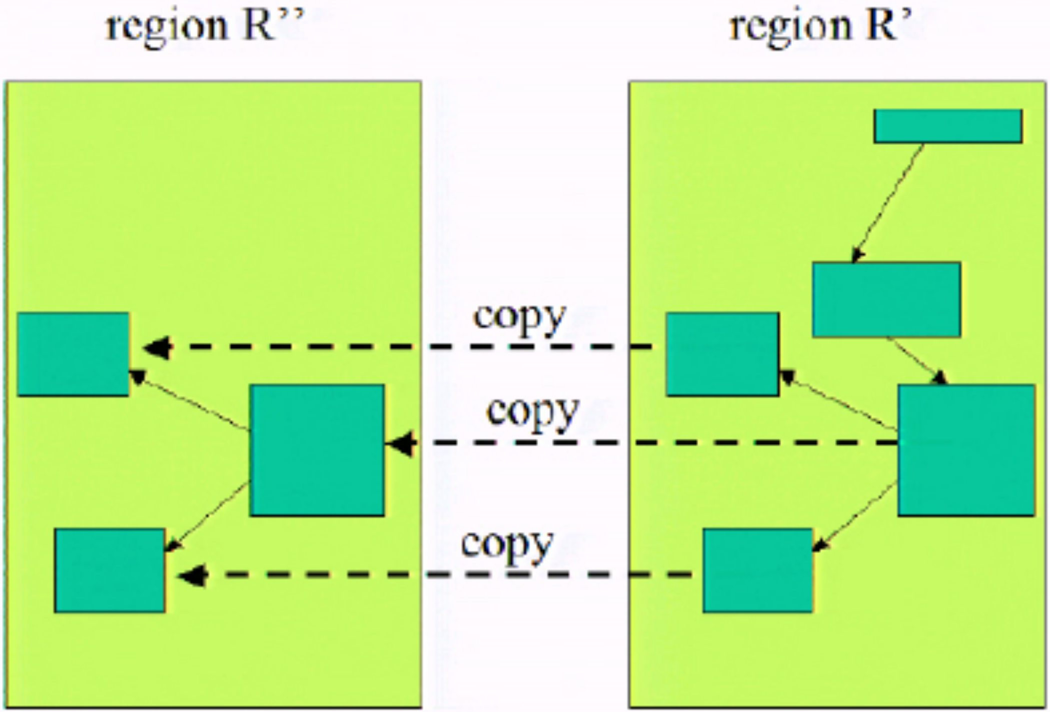


Regions → Copying GC

region R'

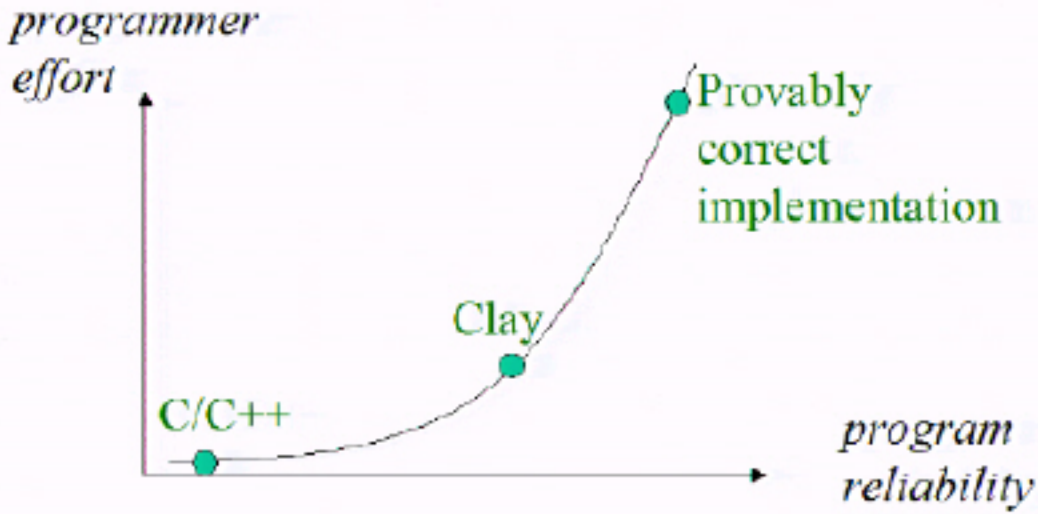


Regions → Copying GC



How Low
Should Safe Languages Go?

How Low Should Safe Languages Go?



Related work

- Vault
- Cyclone
- Extended static checking
- Devil
- Metal